Revising Distributed UNITY Programs is NP-Complete*

Borzoo Bonakdarpour Sandeep S. Kulkarni
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
Email: {borzoo,sandeep}@cse.msu.edu

Abstract

We focus on automated revision techniques for adding UNITY *properties to distributed programs. We show that unlike centralized programs where multiple safety properties and one progress property can be added in polynomial-time, addition of a safety or a progress* UNITY *property to distributed programs is significantly more difficult. Precisely, we show that such addition is NP-complete in the size of the given program's state space. We also propose an efficient symbolic heuristic for addition of a* leads-to *property to distributed programs, which has applications in automated program synthesis.*

Keywords: UNITY, Distributed programs, Revision, Transformation, Formal methods.

1 Introduction

Program correctness is an important aspect and application of formal methods. Designing programs to be *correct-by-construction* is, therefore, highly valuable. Taking the paradigm of correct-by-construction to extreme leads us to synthesizing programs from their specification. While synthesis from specification is undoubtedly useful, it suffers from lack of *reuse*, limitation of expressibility of specification used during synthesis (e.g., in case of undecidable or highly complex languages), and inability to utilize *human knowledge* (e.g., domain expertise). Alternatively, in *program revision* one can transform an input program into an output program that meets additional properties. As a matter of fact, in practice, such properties are frequently identified during a system's life cycle due to reasons such incomplete specification, change of environment, etc. As a concrete example, consider the case where a program is diagnosed with a failed property by a model checker. In such a case, access to automated methods that revise the program with respect to the failed property is highly advantageous. Clearly, transformational approaches that provide reuse allows human expertise to be used in the design of input program, and permits use of expressive specifications during the design of the input program. Inevitably, for such revision to be useful, in addition to satisfaction of new properties, the output program must preserve existing properties of the input program as well.

In our previous work in this context [8], we focused on revising programs with respect to UNITY [7] properties of a high atomicity (*centralized*) program where the program could read and write all program variables in one atomic step. We emphasize that, our revision method in [8] ensures that during revision, satisfaction of all existing UNITY properties of the input program is preserved. In particular, we showed that adding a conjunction of UNITY *safety* properties (i.e., unless, stable, and invariant) and one *progress* property (i.e., leads-to and ensures) can be achieved in polynomial-time. However, we showed that the problem becomes NP-complete if we consider addition of two progress properties. The reason for our focus on UNITY properties is due to the fact that UNITY properties have been found highly valuable in describing a large class of programs.

In this paper, we shift our focus to distributed programs where processes can read and write only a subset of program variables. We expect the concept of program revision to play a more crucial role in the context of distributed programs due to the complex structure of distributed programs where non-determinism and race conditions make it significantly difficult to assert program correctness. We

*This work was partially sponsored by NSF CAREER CCR-0092724 and ONR Grant N00014-01-1-0744.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **2008**	2. REPORT TYPE	3. DATES COVERED **00-00-2008 to 00-00-2008**

4. TITLE AND SUBTITLE **Revising Distributed UNITY Programs is NP-Complete**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Michigan State University,Department of Computer Science and Engineering,East Lansing,MI,48824**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
We focus on automated revision techniques for adding Unity properties to distributed programs. We show that unlike centralized programs where multiple safety properties and one progress property can be added in polynomial-time, addition of a safety or a progress Unity property to distributed programs is significantly more difficult. Precisely, we show that such addition is NP-complete in the size of the given program’s state space. We also propose an efficient symbolic heuristic for addition of a leads-to property to distributed programs, which has applications in automated program synthesis.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**13**	

find somewhat unexpected results about the complexity of adding UNITY properties to distributed programs. In particular, we find that the problem of adding only one UNITY safety property or progress progress property to distributed programs is NP-complete in the size of the input program's state space, even though the corresponding problem can be solved in in polynomial-time for centralized programs.

The knowledge of these complexity bounds is especially important in building tools for incremental synthesis. In particular, the NP-completeness results demonstrate that tools for revising programs must utilize efficient heuristics to expedite the revision algorithm at the cost of *completeness* of that algorithm. With this motivation, in this paper, we propose an efficient symbolic (BDD-based) heuristic that adds a leads-to property to a distributed program. We integrate this heuristic with our tool SYCRAFT [6] that is designed for adding fault-tolerance to existing distributed programs. Leads-to properties are of special interest in fault-tolerant computing where *recovery* within a finite number of steps is essential. To this end, one can first augment the program with all possible recovery transitions that it can use. Clearly, this augmented program does not guarantee that it would recover to a set of legitimate states (e.g., an invariant predicate) although there is a potential to reach the legitimate states from states reached in the presence of faults. In particular, it may continue to execute on a cycle that is entirely outside the legitimate states although from each state there is a path to reach the legitimate states. We apply our heuristics for adding a leads-to property to modify the augmented program so that from any state reached in the presence of faults, the program is guaranteed recovery to its legitimate states within a finite number of steps. As a side effect of the tool for adding leads-to property, we also implement a cycle resolution algorithm. Our experimental results show that this algorithm can also be integrated with existing state-of-the-art model checkers for assisting in developing programs that are correct-by-construction.

Organization. The rest of the paper is organized as follows. In Section 2, we present the preliminary concepts. Then, we formally state the revision problem in Section 3. Section 4 is dedicated to complexity analysis of addition of UNITY safety properties to distributed programs. In Section 5, we present our results on the complexity of addition of UNITY progress properties. We also present our symbolic heuristic and experimental results in Section 5. Related work is discussed in Section 6. We conclude in Section 7. Appendix A provides a summary of notations.

2 Preliminary Concepts

In this section, we formally define the notion of distributed programs. We also reiterate the concept of UNITY properties introduced by Chandy and Misra [7].

2.1 Distributed Programs

Intuitively, we define a distributed program in terms of a set of processes. Each process is in turn specified by a state-transition system and is constrained by some read/write restriction over its set of variables.

Let $V = \{v_0, v_1 \cdots v_n\}$ be a finite set of variables with finite domains $D_0, D_1 \cdots D_n$, respectively. A *state*, say s, is determined by mapping each variable v_i in V, $0 \leq i \leq n$, to a value in D_i. We denote the value of a variable v in state s by $v(s)$. The set of all possible states obtained by variables in V is called the *state space* and is denoted by $\mathcal{S}$. A *transition* is a pair of states of the form (s_0, s_1) where $s_0, s_1 \in \mathcal{S}$.

Definition 2.1 **(state predicate)** Let $\mathcal{S}$ be the state space obtained from variables in V. A *state predicate* is a subset of $\mathcal{S}$. ∎

Definition 2.2 **(transition predicate)** Let $\mathcal{S}$ be the state space obtained from variables in V. A *transition predicate* is a subset of $\mathcal{S} \times \mathcal{S}$. ∎

Definition 2.3 **(process)** A process p is specified by the tuple $\langle V_p, T_p, R_p, W_p \rangle$ where V_p is a set of variables, T_p is a transition predicate in the state space of p (denoted $\mathcal{S}_p$), R_p is a set of variables that p can read, and W_p is a set of variables that p can write such that $W_p \subseteq R_p \subseteq V_p$ (i.e., we assume that p cannot blindly write a variable). ∎

Write restrictions. Let $p = \langle V_p, T_p, R_p, W_p \rangle$ be a process. Clearly, T_p must be disjoint from the following transition predicate due to inability of p to change the value of variables that p cannot write:

$$NW_p = \{(s_0, s_1) \mid v(s_0) \neq v(s_1) \text{ where } v \notin W_p\}.$$

Read restrictions. Let $p = \langle V_p, T_p, R_p, W_p \rangle$ be a process, v be a variable in V_p, and $(s_0, s_1) \in T_p$ where $s_0 \neq s_1$. If v is not in R_p, then p must include a corresponding transition from all states s'_0 where

s_0' and s_0 differ only in the value of v. Let (s_0', s_1') be one such transition. Now, it must be the case that s_1 and s_1' are identical except for the value of v, and, the value of v must be the same in s_0' and s_1'. For instance, let $V_p = \{a, b\}$ and $R_p = \{a\}$. Thus, since p cannot read b, the transition $([a = 0, b = 0], [a = 1, b = 0])$ and the transition $([a = 0, b = 1], [a = 1, b = 1])$ have the same effect as far as p is concerned. Thus, each transition (s_0, s_1) in T_p is associated with the following *group predicate*:

$$
\begin{aligned}
&\mathit{Group}_p(s_0, s_1) = \{(s_0', s_1') \mid \\
&\quad (\forall v \notin R_p \;:\; (v(s_0) = v(s_1) \;\wedge\; v(s_0') = v(s_1'))) \wedge \\
&\quad (\forall v \in R_p \;:\; (v(s_0) = v(s_0') \;\wedge\; v(s_1) = v(s_1')))\}.
\end{aligned}
$$

Definition 2.4 **(distributed program)** A *distributed program* Π is specified by the tuple $\langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ where $\mathcal{P}_\Pi$ is a set of processes and $\mathcal{I}_\Pi$ is a set of initial states. Without loss of generality, we assume that the state space of all processes in $\mathcal{P}_\Pi$ is identical (i.e., $\forall p, q \in \mathcal{P}_\Pi :: (V_p = V_q) \wedge (D_p = D_q)$). Thus, the set of variables (denoted V_Π) and state space of program Π (denoted $\mathcal{S}_\Pi$) are identical to the set of variables and state space of processes of Π, respectively. In this sense, the set $\mathcal{I}_\Pi$ of initial states of Π is a subset of $\mathcal{S}_\Pi$. ∎

Notation. Let $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ be a distributed program (or simply a program). The set $\mathcal{T}_\Pi$ denotes the collection of transition predicates of all processes of Π, i.e., $\mathcal{T}_\Pi = \bigcup_{p \in \mathcal{P}_\Pi} T_p$.

Definition 2.5 **(computation)** Let $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ be a program. A sequence of states, $\overline{s} = \langle s_0, s_1 \cdots \rangle$, is a *computation* of Π iff the following three conditions are satisfied: (1) $s_0 \in \mathcal{I}_\Pi$, (2) $\forall i \geq 0 : (s_i, s_{i+1}) \in \mathcal{T}_\Pi$, and (3) if $\overline{s}$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \mathcal{T}_\Pi$. ∎

For a distributed program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$, we say that a sequence of states, $\overline{s} = \langle s_0, s_1 \cdots s_n \rangle$, is a *computation prefix* of Π iff $\forall j \mid 0 \leq j < n : (s_j, s_{j+1}) \in \mathcal{T}_\Pi$. We distinguish between a *terminating* computation and a *deadlocked* computation. Precisely, when a computation $\overline{s}$ *terminates* in state s_l, we assume that the transition (s_l, s_l) appears in transition predicate of some process in $\mathcal{P}_\Pi$, i.e., $\overline{s}$ can be extended to an infinite computation by stuttering at s_l. On the other hand, if there exists a state s_d such that an outgoing transition (or a self-loop) from s_d appears in transition predicate of no process in $\mathcal{P}_\Pi$ then s_d is a *deadlock* state and a computation of Π that reaches s_d is a *deadlocked computation*. Clearly, such computations cannot be extended to an infinite computation.

2.2 UNITY Properties

We now present the formal definitions for the UNITY properties introduced by Chandy and Misra [7]. UNITY properties are categorized by two classes of *safety* and *progress* properties. These properties are defined next.

Definition 2.6 **(UNITY safety properties)** Let P and Q be arbitrary state predicates.

- (Unless) An infinite sequence of states $\overline{s} = \langle s_0, s_1 \cdots \rangle$ satisfies 'P unless Q' iff $\forall i \geq 0 : (s_i \in (P \cap \neg Q)) \Rightarrow (s_{i+1} \in (P \cup Q))$. Intuitively, if P holds in a state of $\overline{s}$ then either (1) Q never holds in $\overline{s}$ and P is continuously true, or (2) Q becomes true and P holds at least until Q becomes true.

- (Stable) An infinite sequence of states $\overline{s} = \langle s_0, s_1 \cdots \rangle$ satisfies 'stable P' iff $\overline{s}$ satisfies P unless *false*. Intuitively, P is stable iff once it becomes true, it remains true forever.

- (Invariant) An infinite sequence of states $\overline{s} = \langle s_0, s_1 \cdots \rangle$ satisfies 'invariant P' iff $s_0 \in P$ and $\overline{s}$ satisfies stable P. An invariant property always holds. ∎

Definition 2.7 **(UNITY progress properties)** Let P and Q be arbitrary state predicates.

- (Leads-to) An infinite sequence of states $\overline{s} = \langle s_0, s_1 \cdots \rangle$ satisfies 'P leads-to Q' iff $(\forall i \geq 0 : (s_i \in P) \Rightarrow (\exists j \geq i : s_j \in Q))$. In other words, if P holds in a state s_i, $i \geq 0$, of $\overline{s}$ then there exists a state s_j in $\overline{s}$, $i \leq j$, such that Q holds.

- (Ensures) An infinite sequence of states $\overline{s} = \langle s_0, s_1 \cdots \rangle$ satisfies 'P ensures Q' iff (1) if $P \cap \neg Q$ is true in a state s_i, $i \geq 0$, then (1) $s_{i+1} \in (P \cup Q)$, and (2) $\exists j \geq i : s_j \in Q$. In other words, there exists a state s_j where Q eventually becomes true in s_j and P remains true everywhere in between s_i and s_j. ∎

We now define what it means for a program to refine a UNITY property. Note that throughout this paper, we assume that a program and its properties have identical state space.

Definition 2.8 (refines) Let $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ be a program and $\mathcal{L}$ be a UNITY property. We say that Π refines $\mathcal{L}$ iff all computations of Π are infinite and satisfy $\mathcal{L}$. ∎

Definition 2.9 (specification) A UNITY *specification* Σ is the conjunction $\bigwedge_{i=1}^{n} \mathcal{L}_i$ where each $\mathcal{L}_i$ is a UNITY safety or progress property. ∎

One can easily extend the notion of refinement to UNITY specifications as follows. Given a program Π and a specification $\Sigma = \bigwedge_{i=1}^{n} \mathcal{L}_i$, we say that Π refines Σ iff for all i, $1 \leq i \leq n$, Π refines $\mathcal{L}_i$.

Concise representation of safety properties. Notice that the UNITY safety properties can be characterized in terms of a set of *bad transitions* that should never occur in a program computation. For example, $\mathsf{stable}\ P$ requires that a transition, say (s_0, s_1), where $s_0 \in P$ and $s_1 \notin P$, should never occur in any computation of a program that refines $\mathsf{stable}\ P$. Hence, for simplicity, in this paper, when dealing with safety UNITY properties of a program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$, we assume that they are represented by a transition predicate $\mathcal{B} \subseteq \mathcal{S}_\Pi \times \mathcal{S}_\Pi$ whose transitions should never occur in any computation.

3 Problem Statement

Given are a program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ and a (new) UNITY specification Σ_n. Our goal is to devise an automated method which revises Π so that the revised program (denoted $\Pi' = \langle \mathcal{P}_{\Pi'}, \mathcal{I}_{\Pi'} \rangle$) (1) refines Σ_n, and (2) continues refining its existing UNITY specification Σ_e, where Σ_e is unknown. Thus, during the revision, we only want to reuse the correctness of Π with respect to Σ_e so that the correctness of Π' with respect to Σ_e is derived from 'Π refines Σ_e'.

Intuitively, in order to ensure that the revised program Π' continues refining the existing specification Σ_e, we constrain the revision problem so that the set of computations of Π' is a subset of the set of computations of Π. In this sense, since UNITY properties are not existentially quantified (unlike in CTL), we are guaranteed that all computations of Π' satisfy the UNITY properties that participate in Σ_e.

Now, we formally identify constraints on $\mathcal{S}_{\Pi'}$, $\mathcal{I}_{\Pi'}$, and $\mathcal{T}_{\Pi'}$. Observe that if $\mathcal{S}_{\Pi'}$ contains states that are not in $\mathcal{S}_\Pi$, there is no guarantee that the correctness of Π with respect to Σ_e can be reused to ensure that Π' refines Σ_e. Also, since $\mathcal{S}_\Pi$ denotes the set of all states (not just reachable states) of Π, removing states from $\mathcal{S}_\Pi$ is not advantageous. Likewise, $\mathcal{I}_{\Pi'}$ should not have any states that were not there in $\mathcal{I}_\Pi$. Moreover, since $\mathcal{I}_\Pi$ denotes the set of all initial states of Π, we should preserve them during the revision. Finally, we require that $\mathcal{T}_{\Pi'}$ should be a subset of $\mathcal{T}_\Pi$. Note that not all transitions of $\mathcal{T}_\Pi$ may be preserved in $\mathcal{T}_{\Pi'}$. Hence, we must ensure that Π' does not deadlock. Based on Definition 2.9, if (i) $\mathcal{T}_{\Pi'} \subseteq \mathcal{T}_\Pi$, (ii) Π' does not deadlock, and (iii) Π refines Σ_e, then Π' also refines Σ_e. Thus, the *revision problem* is formally defined as follows:

Problem Statement 3.1 Given a program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ and a UNITY specification Σ_n, identify $\Pi' = \langle \mathcal{P}_{\Pi'}, \mathcal{I}_{\Pi'} \rangle$ such that:

- $(C1)$ $\mathcal{S}_{\Pi'} = \mathcal{S}_\Pi$,
- $(C2)$ $\mathcal{I}_{\Pi'} = \mathcal{I}_\Pi$,
- $(C3)$ $\mathcal{T}_{\Pi'} \subseteq \mathcal{T}_\Pi$, and
- $(C4)$ Π' refines Σ_n. ∎

Note that the requirement of deadlock freedom is not explicitly specified in the above problem statement, as it follows from 'Π' refines Σ_n'. Throughout the paper, we use '*revision* of Π with respect to a specification Σ_n (or property $\mathcal{L}$)' and '*addition* of Σ_n (respectively, $\mathcal{L}$) to Π' interchangeably. In Sections 4 and 5, we present our results on developing automated methods that solve the above revision problem with respect to different types of UNITY properties.

4 Adding UNITY Safety Properties to Distributed Programs

As mentioned in Section 2, UNITY safety properties can be characterized by a transition predicate, say $\mathcal{B}$, whose transitions should occur in no computation of a program. In a centralized setting where programs have no restrictions on reading and writing variables, a program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ can be easily revised with respect to $\mathcal{B}$ by simply (1) removing the transitions in $\mathcal{B}$ from $\mathcal{T}_\Pi$, and (2) making newly created deadlock states unreachable [8].

To the contrary, the above approach is not adequate for a distributed setting, as it is *sound* (i.e., it constructs a correct program), but not *complete* (it may fail to find a solution while there exists one). This is due to the issue of read restrictions in distributed programs, which associates each transition of a process with a group predicate. This notion of grouping makes the revision complex, since a revision algorithm has to examine many combinations to determine which group of transitions must be removed and, hence, what deadlock states need to be

handled. Indeed, we show that the issue of read restrictions changes the class of complexity of the revision problem entirely.

Instance. A distributed program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ and UNITY safety specification Σ_n.

Decision problem. Does there exist a program $\Pi' = \langle \mathcal{P}_{\Pi'}, \mathcal{I}_{\Pi'} \rangle$ such that Π' meets the constraints of Problem Statement 3.1 for the above instance?

We now show that the above decision problem is NP-complete by a reduction from the well-known *satisfiability* problem. The SAT problem is as follows:

> Let $x_1, x_2 \cdots x_N$ be propositional *variables*. Given a Boolean formula $y = y_{N+1} \wedge y_{N+2} \cdots y_{M+N}$, where each *clause* y_j, $N+1 \leq j \leq M+N$, is a disjunction of three or more literals, does there exist an assignment of truth values to $x_1, x_2 \cdots x_N$ such that y is satisfiable?

We note that the unconventional subscripting of variables and clauses in the above definition of the SAT problem is deliberately chosen to make our proofs simpler.

Theorem 4.1 *The problem of adding a* UNITY *safety property to a distributed program is NP-complete.*

Proof. Since showing membership to NP is straightforward, we only need to show that the problem is NP-hard. Towards this end, we present a polynomial-time mapping from an instance of the SAT problem to a corresponding instance of our revision problem. Thus, we construct $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ as follows.

Variables. The set of variables of program Π and, hence, its processes is $V = \{v_0, v_1, v_2, v_3, v_4\}$. The domain of these variables are respectively as follows: $\{-1, 0, 1\}$, $\{-1, 0, 1\}$, $\{0, 1\}$, $\{0, 1\}$, $\{1, 2 \cdots M + N\} \cup \{j^i \mid (1 \leq i \leq N) \wedge (N+1 \leq j \leq M+N)\}$. We note that j^i in the last set is not an exponent, but a denotational symbol.

Reachable states. The set of reachable states in our mapping are as follows:

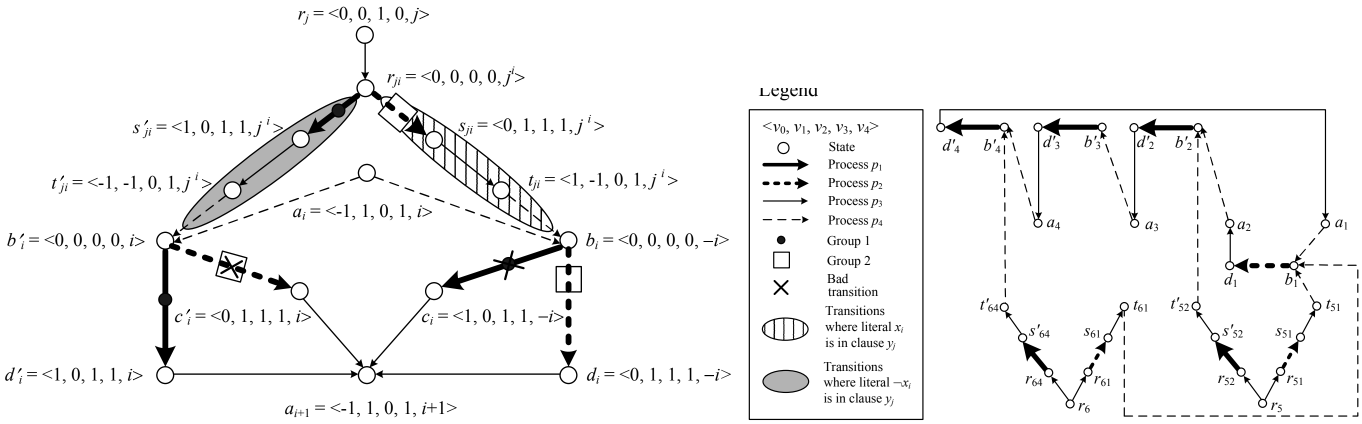
- For each propositional variable x_i, $1 \leq i \leq N$, in the instance of the SAT problem, we introduce the following states (see Figure 1-a): $a_i, b_i, b'_i, c_i, c'_i, d_i, d'_i$. We require that states a_1 and a_{N+1} are identical.
- For each clause y_j, $N+1 \leq j \leq M+N$, we introduce state r_j.
- For each clause y_j, $N+1 \leq j \leq M+N$, and variable x_i in clause y_j, $1 \leq i \leq N$, we introduce the following states: $r_{ji}, s_{ji}, s'_{ji}, t_{ji}, t'_{ji}$.

Value assignments. Assignment of values to each variable at each state is shown in Figure 1-a (denoted by $< v_0, v_1, v_2, v_3, v_4 >$). This part of our mapping is the most crucial factor in forming group predicates.

Processes. Program Π consists of four processes. Formally, $\mathcal{P}_\Pi = \{p_1, p_2, p_3, p_4\}$. Transition predicate and read/write restrictions of processes in $\mathcal{P}_\Pi$ are as follows:

- **Read/write restrictions.** The read/write restrictions of processes p_1, p_2, p_3, and p_4 are as follows:
  - $R_{p_1} = \{v_0, v_2, v_3\}$ and $W_{p_1} = \{v_0, v_2, v_3\}$.
  - $R_{p_2} = \{v_1, v_2, v_3\}$ and $W_{p_2} = \{v_1, v_2, v_3\}$.
  - $R_{p_3} = \{v_0, v_1, v_2, v_3, v_4\}$ and $W_{p_3} = \{v_0, v_1, v_2, v_4\}$.
  - $R_{p_4} = \{v_0, v_1, v_2, v_3, v_4\}$ and $W_{p_4} = \{v_0, v_1, v_3, v_4\}$.
- **Transition predicates.** For each propositional variable x_i, $1 \leq i \leq N$, we include the following transitions in processes p_1, p_2, p_3, and p_4 (see Figure 1-a):
  - $T_{p_1} = \{(b'_i, d'_i), (b_i, c_i) \mid 1 \leq i \leq N\}$.
  - $T_{p_2} = \{(b'_i, c'_i), (b_i, d_i) \mid 1 \leq i \leq N\}$.
  - $T_{p_3} = \{(c'_i, a_{i+1}), (c_i, a_{i+1}), (d'_i, a_{i+1}), (d_i, a_{i+1}) \mid 1 \leq i \leq N\}$.
  - $T_{p_4} = \{(a_i, b_i), (a_i, b'_i) \mid 1 \leq i \leq N\}$.

  Moreover, corresponding to each clause y_j, $N+1 \leq j \leq M+N$, and variable x_i, $1 \leq i \leq N$, in clause y_j, we include transition (r_j, r_{ji}) in T_{p_3} and the following:
  - If x_i is a literal in clause y_j then we include transition (r_{ji}, s_{ji}) in T_{p_2}, (s_{ji}, t_{ji}) in T_{p_3}, and (t_{ji}, b_i) in T_{p_4}.
  - If $\neg x_i$ is a literal in clause y_j then we include transition (r_{ji}, s'_{ji}) in T_{p_1}, (s'_{ji}, t'_{ji}) in T_{p_3}, and (t'_{ji}, b'_i) in T_{p_4}.



(a) Mapping SAT to addition of UNITY safety properties.

(b) The structure of the revised program for Boolean formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$, where $x_1 = true$, $x_2 = false$, $x_3 = false$, and $x_4 = false$.

Figure 1: Reduction from the SAT problem.

Note that only for the sake of illustration, Figure 1-a shows all possible transitions. However, in order to construct Π, based on the existence of x_i or $\neg x_i$ in y_j, we only include a subset of the transitions.

Initial states. The set $\mathcal{I}_\Pi$ represents clauses of the instance of the SAT problem, i.e., $\mathcal{I}_\Pi = \{r_j \mid N+1 \leq j \leq M+N\}$.

Safety property. Let P be a state predicate that contains all reachable states in Figure 1-a except c_i and c'_i (i.e., $c_i, c'_i \in \neg P$). Thus, the properties stable P and invariant P can be characterized by the transition predicate $\mathcal{B} = \{(b_i, c_i), (b'_i, c'_i) \mid 1 \leq i \leq N\}$. Similarly, let P and Q be two state predicates that contain all reachable states in Figure 1-a except c_i and c'_i. Thus, the safety property P unless Q can be characterized by $\mathcal{B}$ as well. In our mapping, we let $\mathcal{B}$ represent the safety specification for which Π has to be revised.

Before we present our reduction from the SAT problem using the above mapping, we make the following observations regarding the grouping of transitions in different processes:

1. Due to inability of process p_1 to read variable v_4, for all i, $1 \leq i \leq N$, transitions $(r_{ji}, s'_{ji}), (b'_i, d'_i)$, and (b_i, c_i) are grouped in p_1.

2. Due to inability of process p_2 to read variable v_4, for all i, $1 \leq i \leq N$, transitions $(r_{ji}, s_{ji}), (b_i, d_i)$, and (b'_i, c'_i) are grouped in p_2.

3. Transitions grouped with the rest of the transitions in Figure 1-a are unreachable and, hence, are irrelevant.

Now, we show that the answer to the SAT problem is affirmative if and only if there exists a solution to the revision problem. Thus, we distinguish two cases:

- ($\Rightarrow$) First, we show that if the given instance of the SAT formula is satisfiable then there exists a solution that meets the requirements of the revision decision problem. Since the SAT formula is satisfiable, there exists an assignment of truth values to all variables x_i, $1 \leq i \leq N$, such that each y_j, $N+1 \leq j \leq M+N$, is true. Now, we identify a program Π', that is obtained by adding the safety property represented by $\mathcal{B}$ to program Π as follows.
  - The state space of Π' consists of all the states of Π, i.e., $\mathcal{S}_\Pi = \mathcal{S}_{\Pi'}$.
  - The initial states of Π' consists of all the initial states of Π, i.e., $\mathcal{I}_\Pi = \mathcal{I}_{\Pi'}$.
  - For each variable x_i, $1 \leq i \leq N$, if x_i is *true* then we include the following transitions: (a_i, b_i) in T_{p_4}, (b_i, d_i) in T_{p_2}, and (d_i, a_{i+1}) in T_{p_3}.
  - For each variable x_i, $1 \leq i \leq N$, if x_i is *false* then we include the following transitions:(a_i, b'_i) in T_{p_4}, (b'_i, d'_i) in T_{p_1}, and (d'_i, a_{i+1}) in T_{p_3}.

– For each clause y_j, $N+1 \leq j \leq M+N$, that contains literal x_i, if x_i is *true*, we include the following transitions: (r_j, r_{ji}) in T_{p_4}, (r_{ji}, s_{ji}) in T_{p_2}, (s_{ji}, t_{ji}) in T_{p_3}, and (t_{ji}, b_i) in T_{p_4}.

– For each clause y_j, $N+1 \leq j \leq M+N$, that contains literal $\neg x_i$, if x_i is *false*, we include the following transitions: (r_j, r_{ji}) in T_{p_4}, (r_{ji}, s'_{ji}) in T_{p_1}, (s'_{ji}, t'_{ji}) in T_{p_3}, and (t'_{ji}, b'_i) in T_{p_4}.

As an illustration, we show the partial structure of Π', for the formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$, where $x_1 = true$, $x_2 = false$, $x_3 = false$, and $x_4 = false$, in Figure 1-b. Notice that states whose all outgoing and incoming transitions are eliminated are not illustrated. Now, we show that Π' meets the requirements of the Problems Statement 3.1:

1. The first three constraints of the decision problem are trivially satisfied by construction.

2. We now show that constraint $C4$ holds. First, it is easy to observe that by construction, there exist no reachable deadlock states in the revised program. Hence, if Π refines UNITY specification Σ_e then Π' refines Σ_e as well. Moreover, if a computation of Π' reaches a state b_i for some i, from an initial state r_j (i.e., x_i is *true* in clause y_j) then that computation cannot violate safety since bad transition (b_i, c_i) is removed. This is due to the fact that (b_i, c_i) is grouped with transition (r_{ji}, s'_{ji}) and this transition is not included in Π', as literal x_i is *true* in y_j. Likewise, if a computation of Π' reaches a state b'_i for some i, from initial state r_j (i.e., x_i is *false* in clause y_j) then that computation cannot violate safety since transition (b'_i, c'_i) is removed. This is due to the fact that (b'_i, c'_i) is grouped with transition (r_{ji}, s_{ji}) and this transition is not included in Π', as x_i is *false*. Thus, Π' refines Σ_n.

- ($\Leftarrow$) Next, we show that if there exists a solution to the revision problem for the instance identified by our mapping from the SAT problem, then the given SAT formula is satisfiable. Let Π' be the program that is obtained by adding the safety property Σ_n to program Π. Now, in order to obtain a solution for SAT, we proceed as follows. If there exists a computation of Π' where state b_i is reachable then we assign x_i the truth value *true*. Otherwise, we assign the truth value *false*.

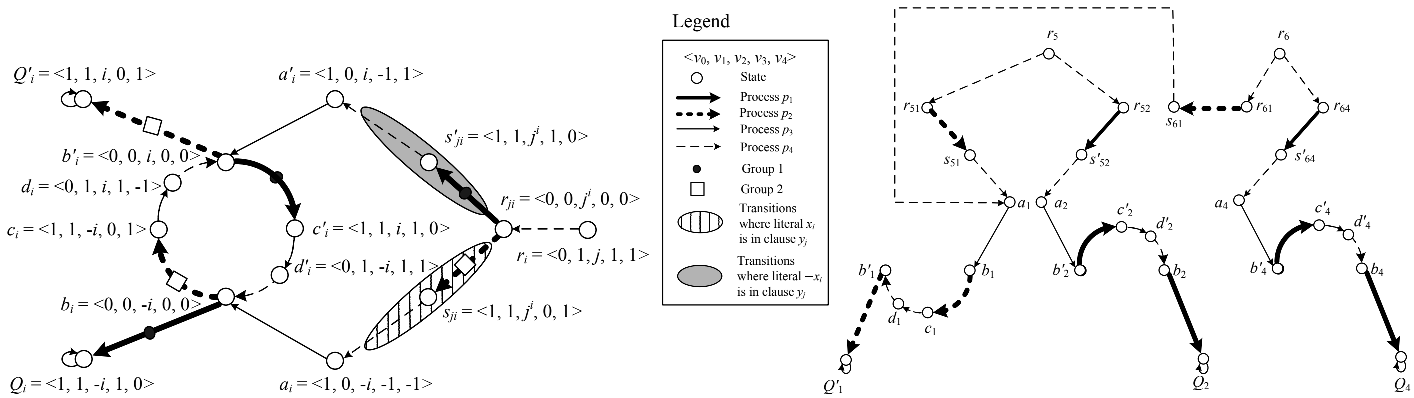
  We now show that the above truth assignment satisfies all clauses. Let y_j be a clause for some j, $N+1 \leq j \leq M+N$, and let r_j be the corresponding initial state in Π'. Since r_j is an initial state and Π' cannot deadlock, the transition (r_j, r_{ji}) must be present in Π', for some i, $1 \leq i \leq N$. By the same argument, there must exist some transition that originates from r_{ji}. This transition terminates in either s_{ji} or s'_{ji}. Observe that Π' cannot have both transitions, as grouping of transitions will include both (b_i, c_i) and (b'_i, c'_i) which in turn causes violation of safety by Π'. Now, if the transition from r_{ji} terminates in s_{ji}, then clause y_j contains literal x_i and x_i is assigned the truth value true. Hence, y_j evaluates to true. Likewise, if the transition from r_{ji} terminates in s'_{ji} then clause y_j contains literal $\neg x_i$ and x_i is assigned the truth value *false*. Hence, y_j evaluates to true. Therefore, the assignment of values considered above is a satisfying truth assignment for the given SAT formula. ∎

5 Adding UNITY Progress Properties to Distributed Programs

This section is organized as follows. In Subsection 5.1, we show that adding a UNITY progress property to a distributed program is NP-complete. Then, in Subsection 5.2, we present a symbolic heuristic for adding a leads-to property to a distributed program.

5.1 Complexity

In a centralized setting, where programs have no restriction on reading and writing variables, a program, say $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$, can be easily revised with respect to a progress property by simply (1) breaking non-progress cycles that prevent a program to eventually reach a desirable state predicate, and (2) removing deadlock states [8]. To the contrary, in a distributed setting, due to the issue of grouping, it matters which transition (and as a result its corresponding group) is removed to break a non-progress cycle.



(a) Mapping SAT to addition of a leads-to property.

(b) The structure of the revised program for Boolean formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$, where $x_1 = \mathit{true}$, $x_2 = \mathit{false}$, $x_3 = \mathit{false}$, and $x_4 = \mathit{false}$.

Figure 2: Reduction from the SAT problem.

Instance. A distributed program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ and UNITY progress property Σ_n.

Decision problem. Does there exist a program $\Pi' = \langle \mathcal{P}_{\Pi'}, \mathcal{I}_{\Pi'} \rangle$ such that Π' meets the constraints of Problem Statement 3.1 for the above instance?

Theorem 5.1 *The problem of adding a* UNITY *progress property to a distributed program is NP-complete.*

Proof. Since showing membership to NP is straightforward, we only show that the problem is NP-hard by a reduction from the SAT problem. First, we present a polynomial-time mapping.

Variables. The set of variables of program Π and, hence, its processes is $V = \{v_0, v_1, v_2, v_3, v_4\}$. The domain of these variables are respectively as follows: $\{0, 1\}$, $\{0, 1\}$, $\{1, 2 \cdots M + N\} \cup \{j^i \mid (1 \leq i \leq N) \wedge (N + 1 \leq j \leq M + N)\}$, $\{-1, 0, 1\}$, and $\{-1, 0, 1\}$.

Reachable states. The set of reachable states in our mapping are as follows:

- For each propositional variable x_i, $1 \leq i \leq N$, we introduce the following states (see Figure 2-a): a_i, a'_i, b_i, b'_i, c_i, c'_i, d_i, d'_i, Q_i, and Q'_i.
- For each clause y_j, $N + 1 \leq j \leq M + N$, we introduce state r_j.
- For each clause y_j, $N + 1 \leq j \leq M + N$, and x_i, $1 \leq i \leq N$, in clause y_j, we introduce states r_{ji}, s_{ji}, and s'_{ji}.

Value assignments. Assignment of values to each variable at each state is shown in Figure 2-a (denoted by $< v_0, v_1, v_2, v_3, v_4 >$).

Processes. Program Π consists of four processes. Formally, $\mathcal{P}_\Pi = \{p_1, p_2, p_3, p_4\}$. Transition predicate and read/write restrictions of processes in $\mathcal{P}_\Pi$ are as follows:

- **Read/write restrictions.** The read/write restrictions of processes p_1, p_2, p_3, and p_4 are as follows:
  - $R_{p_1} = \{v_0, v_1, v_3\}$ and $W_{p_1} = \{v_0, v_1, v_3\}$.
  - $R_{p_2} = \{v_0, v_1, v_4\}$ and $W_{p_2} = \{v_0, v_1, v_4\}$.
  - $R_{p_3} = \{v_0, v_1, v_2, v_3, v_4\}$ and $W_{p_3} = \{v_0, v_2, v_3, v_4\}$.
  - $R_{p_4} = \{v_0, v_1, v_2, v_3, v_4\}$ and $W_{p_4} = \{v_1, v_2, v_3, v_4\}$.
- **Transition predicates.** For each propositional variable x_i, $1 \leq i \leq N$, we include the following transitions in processes p_1, p_2, p_3, and p_4 (see Figure 2-a):
  - $T_{p_1} = \{(b'_i, c'_i), (b_i, Q_i) \mid 1 \leq i \leq N\}$.
  - $T_{p_2} = \{(b_i, c_i), (b'_i, Q'_i) \mid 1 \leq i \leq N\}$.
  - $T_{p_3} = \{(a_i, b_i), (a'_i, b'_i), (c_i, d_i), (c'_i, d'_i), (Q_i, Q_i), (Q'_i, Q'_i) \mid 1 \leq i \leq N\}$.
  - $T_{p_4} = \{(d'_i, b_i), (d_i, b'_i) \mid 1 \leq i \leq N\}$.

Moreover, corresponding to each clause y_j, $N+1 \leq j \leq M+N$, and variable x_i, $1 \leq i \leq N$, in clause y_j, we include transition (r_j, r_{ji}) in T_{p_3} and the following:

- If x_i is a literal in clause y_j then we include transition (r_{ji}, s_{ji}) in T_{p_2}, and (s_{ji}, a_i) in T_{p_4}.
- If $\neg x_i$ is a literal in clause y_j then we include transition (r_{ji}, s'_{ji}) in T_{p_1} and (s'_{ji}, a'_i) in T_{p_4}.

Note that for the sake of illustration Figure 2-a shows all possible transitions. However, in order to construct Π, based on the existence of x_i or $\neg x_i$ in y_j, we only include a subset of transitions.

Initial states. The set $\mathcal{I}_\Pi$ of Π is the set of states that represent clauses of the boolean formula in the instance of SAT, i.e., $\mathcal{I}_\Pi = \{r_j \mid N{+}1 \leq j \leq M{+}N\}$.
Progress property. In our mapping, the desirable progress property is of the form $\Sigma_n \equiv$ (*true* leads-to Q), where $Q = \{Q_i, Q'_i \mid 1 \leq i \leq N\}$ (see Figure 2-a). Observe that Σ_n is a leads-to as well as an ensures property. This property in Linear Temporal Logic (LTL) is denoted by $\Box\Diamond Q$ (called *always eventually* Q).

Before we present our reduction from the SAT problem using the above mapping, we make the following observations regarding the grouping of transitions in different processes:

1. Due to inability of process p_1 to read variable v_2, for all i, $1 \leq i \leq N$, transitions (r_{ji}, s'_{ji}), (b'_i, c'_i), and (b_i, Q_i) are grouped in process p_1.

2. Due to inability of process p_2 to read variable v_2, for all i, $1 \leq i \leq N$, transitions (r_{ji}, s_{ji}), (b_i, c_i), and (b'_i, Q'_i) are grouped in process p_2.

3. Transitions grouped with the rest of the transitions in Figure 2-a are unreachable and, hence, are irrelevant.

We distinguish the following two cases for reducing the SAT problem to our revision problem :

- ($\Rightarrow$) First, we show that if the given instance of the SAT formula is satisfiable then there exists a solution that meets the requirements of the revision decision problem. Since the SAT formula is satisfiable, there exists an assignment of truth values to all variables x_i, $1 \leq i \leq N$, such that each y_j, $N+1 \leq j \leq M+N$, is true. Now, we identify a program Π', that is obtained by adding the progress property $\Box\Diamond Q$ to program Π as follows.

  - The state space of Π' consists of all the states of Π, i.e., $\mathcal{S}_\Pi = \mathcal{S}_{\Pi'}$.
  - The initial states of Π' consists of all the initial states of Π, i.e., $\mathcal{I}_\Pi = \mathcal{I}_{\Pi'}$.
  - For each variable x_i, $1 \leq i \leq N$, if x_i is *true* then we include the following transitions: (a_i, b_i), (c_i, d_i), and (Q'_i, Q'_i) in T_{p_3}, (b_i, c_i) and (b'_i, Q'_i) in T_{p_2}, and, (d_i, b'_i) in T_{p_4}.
  - For each variable x_i, $1 \leq i \leq N$, if x_i is *false* then we include the following transitions: (a'_i, b'_i), (c'_i, d'_i), and (Q_i, Q_i) in T_{p_3}, (b'_i, c'_i) and (b_i, Q_i) in T_{p_1}, and, (d'_i, b_i) in T_{p_4}.
  - For each clause y_j, $N+1 \leq j \leq M+N$, that contains literal x_i, if x_i is *true*, we include transition (r_j, r_{ji}) in T_{p_4}, (r_{ji}, s_{ji}) in T_{p_2}, and, (s_{ji}, a_i) in T_{p_4}.
  - For each clause y_j, $N+1 \leq j \leq M+N$, that contains literal $\neg x_i$, if x_i is *false*, we include transition (r_j, r_{ji}) in T_{p_4}, (r_{ji}, s'_{ji}) in T_{p_1}, and, (s'_{ji}, a'_i) in T_{p_4}.

  As an illustration, we show the partial structure of Π', for the formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$, where $x_1 = $ *true*, $x_2 = $ *false*, $x_3 = $ *false*, and $x_4 = $ *false* in Figure 2-b. Notice that states whose all outgoing and incoming transitions are eliminated are not illustrated. Now, we show that Π' meets the requirements of the Problems Statement 3.1:

  1. The first three constraints of the decision problem are trivially satisfied by construction.
  2. We now show that constraint $C4$ holds. First, it is easy to observe that by construction, there exist no reachable deadlock states in the revised program. Hence, if Π refines UNITY specification Σ_e then Π' refines Σ_e as well. Moreover, by construction, all computations of Π' eventually reach either Q_i or Q'_i and will stutter there. This is due to the fact that if literal x_i is *true* in clause y_j then transition (r_{ji}, s'_{ji}) is not included in Π' and, hence,

its group-mates (b'_i, c'_i) and (b_i, Q_i) are not in $\mathcal{T}_{\Pi'}$ as well. Consequently, a computation that starts from r_j eventually reaches Q'_i. Likewise, if literal $\neg x_i$ is *false* in clause y_j then transition (r_{ji}, s_{ji}) is not included in Π' and, hence, its group-mates (b_i, c_i) and (b'_i, Q'_i) are not in $\mathcal{T}_{\Pi'}$ as well. Consequently, a computation that starts from r_j eventually reaches Q_i. Hence, Π' refines $\Sigma_n \equiv \Box\Diamond Q$.

- ($\Leftarrow$) Next, we show that if there exists a solution to the revision problem for the instance identified by our mapping from the SAT problem, then the given SAT formula is satisfiable. Let Π' be the program that is obtained by adding the progress property in $\Sigma_n \equiv \Box\Diamond Q$ to program Π. Now, in order to obtain a solution for SAT, we proceed as follows. If there exists a computation of Π' where state a_i is reachable then we assign x_i the truth value *true*. Otherwise, we assign the truth value *false*.

  We now show that the above truth assignment satisfies all clauses. Let y_j be a clause for some j, $N+1 \le j \le M+N$, and let r_j be the corresponding initial state in Π'. Since r_j is an initial state and Π' cannot deadlock, the transition (r_j, r_{ji}) must be present in Π', for some i, $1 \le i \le N$. By the same argument, there must exist some transition that originates from r_{ji}. This transition terminates in either s_{ji} or s'_{ji}. Observe that Π' cannot have both transitions, as grouping of transitions will include transitions (b_i, c_i) and (b'_i, c'_i). If this is the case, Π' does not refine the property $\Box\Diamond Q$ due to the existence of cycle $b_i \to c_i \to d_i \to b'_i \to c'_i \to d'_i \to b_i$. Thus, there can be one and only one outgoing transition from r_{ji} in Π'. Now, if the transition from r_{ji} terminates in s_{ji}, then clause y_j contains literal x_i and x_i is assigned the truth value true. Hence, y_j evaluates to true. Likewise, if the transition from r_{ji} terminates in s'_{ji} then clause y_j contains literal $\neg x_i$ and x_i is assigned the truth value *false*. Hence, y_j evaluates to true. Therefore, the assignment of values considered above is a satisfying truth assignment for the given SAT formula. ∎

5.2 A Symbolic Heuristic for Adding Leads-To Properties

We now present a BDD-based heuristic for adding leads-to properties to distributed programs due to its interesting applications in automated addition of recovery for synthesizing fault-tolerant distributed programs.

The NP-hardness reduction presented in the proof of Theorem 5.1 precisely shows where the complexity of the problem lies in. Indeed, Figure 2-a shows that transition (b_i, c_i) (respectively, (b'_i, c'_i)), which can potentially be removed to break the non-progress cycle $b_i \to c_i \to d_i \to b'_i \to c'_i \to d'_i \to b_i$ is grouped with the critical transition (r_{ji}, s_{ji}) (respectively, (r_{ji}, s'_{ji})) which ensures state r_{ji} and consequently initial state r_j are not deadlock states. Thus, a heuristic that adds a leads-to property to a distributed program needs to address this issue.

Our heuristic works as follows (cf. Figure 3-a). The Algorithm Add_LeadsTo takes a distributed program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ and a property P leads-to Q as input, where P and Q are two arbitrary state predicates in the state space of Π. The algorithm (if successful) returns transition predicate of the derived program $\Pi' = \langle \mathcal{P}_{\Pi'}, \mathcal{I}_{\Pi'} \rangle$ that refines P leads-to Q as output. In order to transform Π to Π', first, the algorithm ranks states that can be reached from P based on the length of their shortest path to Q (Line 2). Then, it attempts to break non-progress cycles (Lines 3-13). To this end, it first computes the set of cycles that are reachable from P (Line 4). This computation can be accomplished using any BDD-based cycle detection algorithm. We apply the Emerson-Lie method [10]. Then, the algorithm removes transitions that participate in a cycle and whose rank of source state is less than or equal to the rank of destination state (Lines 6-10). However, since removal of a transition must take place with its entire group predicate, we do not remove a transition that causes creation of deadlock states in Q. Instead, we make the corresponding cycle unreachable (Line 8). This can be done by simply removing transitions that terminate in a state on the cycle. Thus, if removal of a group of transitions does not create new deadlock states in Q, the algorithm removes them (Line 10). Finally, since removal of transitions may create deadlock states outside Q but reachable from P, we need to eliminate those deadlock states (Line 15). Such elimination can be accomplished using the BDD-based method proposed in [5].

Algorithm 1 Add_LeadsTo

Input: A distributed program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ and property P leads-to Q.
Output: If successful, transition predicate $\mathcal{T}_{\Pi'}$ of the new program.

```
1: repeat
2:     Let Rank[i] contain the state predicate whose length of shortest path
       to Q is i, where Rank[0] = Q and Rank[∞] = the state predicate that
       is reachable from P, but cannot reach Q;
3:     for all i and j do
4:         C := ComputeCycles(T_Π, P);
5:         if (i ≤ j) ∧ (i ≠ 0) ∧ (i ≠ ∞) then
6:             tmp := Group(⟨C ∧ Rank[i]⟩ ∧ ⟨C ∧ Rank[j]⟩′);
7:             if removal of tmp from T_Π eliminates a state from Q then
8:                 Make ⟨C ∧ tmp⟩ unreachable
9:             else
10:                T_Π := T_Π − tmp;
11:            end if
12:        end if
13:    end for
14: until Rank[∞] = {}
15: T_Π′ := EliminateDeadlockStates(P, Q, ⟨P_Π, I_Π⟩);
16: return T_Π′;
```

(a) Symbolic heuristic

	Space		*Time(s)*		
	reachable states	**memory (KB)**	**cycle detection**	**pruning transitions**	**total**
BA^5	10^4	12	0.5	2.5	3
BA^{10}	10^8	18	5	18	23
BA^{15}	10^{12}	26	47	76	125
BA^{20}	10^{16}	29	522	372	894
BA^{25}	10^{20}	30	3722	1131	4853
TR^5	10^2	6	0.2	0.3	0.5
TR^{10}	10^5	7	13	2	15
TR^{15}	10^7	10	470	10	480
TR^{20}	10^9	33	2743	173	2916
TR^{25}	10^{11}	53	22107	2275	24382

(b) Experimental results

Figure 3: Adding leads-to property to distributed programs.

Given $O(n^2)$ complexity of the cycle detection algorithm [10], it is straightforward to observe that the complexity of our heuristic is $O(n^4)$, where n is the size of state space of Π. In order to evaluate the performance of our heuristic, we have implemented the Algorithm Add_LeadsTo in our tool SYCRAFT [6]. This heuristic can be used for adding *recovery* in order to synthesize fault-tolerant distributed programs by performing the following two steps. First, we add all possible transitions that start from *fault-span* predicate T (i.e., set of all reachable states in the presence of faults) and end in T. Then, we apply the Algorithm Add_LeadsTo for property $(T - S)$ leads-to S, where S is a set of legitimate states (i.e., an invariant predicate).

Figure 3-b illustrates experimental results of our heuristic for adding such recovery. All experiments are run on a PC with a 2.8GHz Intel Xeon processor and 1.2GB RAM. The BDD representation of the Boolean formulae has been done using the Glu/CUDD package [18]. Our experiments target addition of recovery two well-known problems in fault-tolerant distributed computing, namely, the *Byzantine agreement* problem [14] (denote BA^i) and the *token ring* problem [2] (denoted TR^i), where i is the number of processes. Figure 3-b shows the size of reachable states in the presence of faults, memory usage, total time spent to add the desirable leads-to property, time spent for cycle detection (i.e., Line 4 in Figure 3-a), and time spent for pruning transitions that participate in a cycle. Given the huge size of state space and complexity of structure of programs in our experiments, we find the experimental results quite encouraging. We note that the reason that TR and BA behave differently as their number of processes grow is due to their different structures, existing cycles, and number of reachable states. In particular, the state space of TR is highly reachable and its original program has a cycle that includes all of its legitimate states, which is not the case for BA. We also note that in case of TR, the symbolic heuristic presented in this subsection tend to be slower than the constructive layered approach introduced in [5]. However, the approach in this paper is more general and has a better potential of success than the approach in [5].

6 Related Work

The most relevant work to this paper proposes automated transformation techniques for adding UNITY properties to centralized programs [8]. We showed that addition of multiple UNITY safety properties along with a single progress property to a centralized program can be accomplished in polynomial-time. We also showed that the problem of simultaneous addition of two leads-to properties to a centralized program is NP-complete.

Existing synthesis methods in the literature mostly focus on deriving the synchronization skeleton of a program from its specification (expressed in terms of temporal logic expressions or finite-state automata) [1, 3, 4, 9, 15–17]. Although such synthe-

sis methods may have differences with respect to the input specification language and the program model that they synthesize, the general approach is based on the satisfiability proof of the specification. This makes it difficult to provide *reuse* in the synthesis of programs; i.e., any changes in the specification require the synthesis to be restarted from scratch.

Algorithms for automatic addition of fault-tolerance to distributed programs are studied from different perspectives [5,11–13]. These (enumerative and symbolic) algorithms add fault-tolerance concerns to existing programs in the presence of faults, and guarantee not to add new behaviors to that program in the absence of faults. Most problems in addition of fault-tolerance to distributed programs are known to NP-complete. Thus, in this paper, we find it somewhat unexpected that corresponding problems in the absence of faults remain NP-complete.

7 Conclusion and Future Work

In this paper, we concentrated on automated techniques for *revising* distributed programs with respect to Unity properties. We showed that unlike centralized programs where multiple Unity safety properties along with one progress property can be added in polynomial-time [8], the problem is NP-complete for distributed programs. We also introduced and implemented a BDD-based heuristic for adding a leads-to property to distributed programs in our tool Sycraft [6]. Our experiments show encouraging results paving the path for applying automated techniques for deriving programs that are *correct-by-construction* in practice.

For future work, we plan to identify sub-problems where one can devise sound and complete algorithms that add Unity properties to distributed programs in polynomial-time. We also plan to devise heuristics for adding other types of Unity properties to distributed programs.

References

[1] A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. In *Principles of Distributed Computing (PODC)*, pages 173–182, 1998.

[2] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, 1998.

[3] P. C. Attie. Synthesis of large concurrent programs via pairwise composition. In *International Conference on Concurrency Theory (CONCUR)*, pages 130–145, London, UK, 1999. Springer-Verlag.

[4] P.C. Attie and E. A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(2):187 – 242, 2001.

[5] B. Bonakdarpour and S. S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, 2007.

[6] B. Bonakdarpour and S. S. Kulkarni. SYCRAFT: A tool for synthesizing fault-tolerant distributed programs. In *Concurrency Theory (CONCUR)*, pages 167–171, 2008.

[7] K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[8] A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour. Revising UNITY programs: Possibilities and limitations. In *International Conference on Principles of Distributed Systems (OPODIS)*, LNCS 3974, pages 275–290, 2005.

[9] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[10] E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional model mu-calculus. In *Logic in Computer Science (LICS)*, pages 267–278, 1986.

[11] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 82–93, 2000.

[12] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 130–140, 2001.

[13] S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *International Conference on Distributed Computing Systems*, 2002.

[14] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[15] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):68–93, 1984.

[16] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Principles of Programming Languages (POPL)*, pages 179–190, 1989.

[17] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *International Colloquium on Automata, Languages, and Programming*, number 372 in Lecture Notes in Computer Science, pages 652–671. Springer-Verlag, 1989.

[18] F. Somenzi. CUDD: Colorado University Decision Diagram Package. `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.

Appendix

A Summary of Notations

V	set of variables
D	domain of variables
s	state
$\mathcal{S}$	state space
T_p	transition predicate of process p
W_p	set of variables that process p can write
R_p	set of variables that process p can read
Π	distributed program
$\mathcal{P}_\Pi$	processes of program Π
$\mathcal{I}_\Pi$	initial states of program Π
$\mathcal{T}_\Pi$	transition predicate of program Π
P, Q	state predicates
$\overline{s}$	computation
$\mathcal{L}$	UNITY property
Σ_e	existing specification
Σ_n	new specification
$\mathcal{B}$	transition predicate that characterizes a safety UNITY property
$\Box\Diamond Q$	always eventually Q